# *THE VERSATILE ACTOR:* TOWARD COMPOSITIONAL PROGRAMMING OF DISTRIBUTED APPLICATIONS

PsiEta 2010

John Field, IBM Research

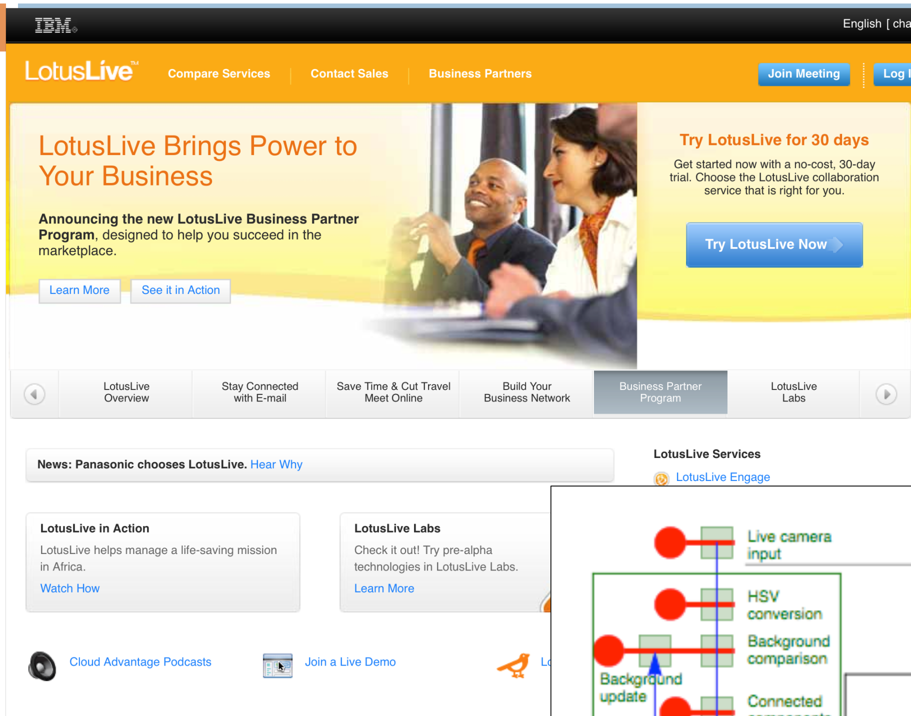
# Collaborators

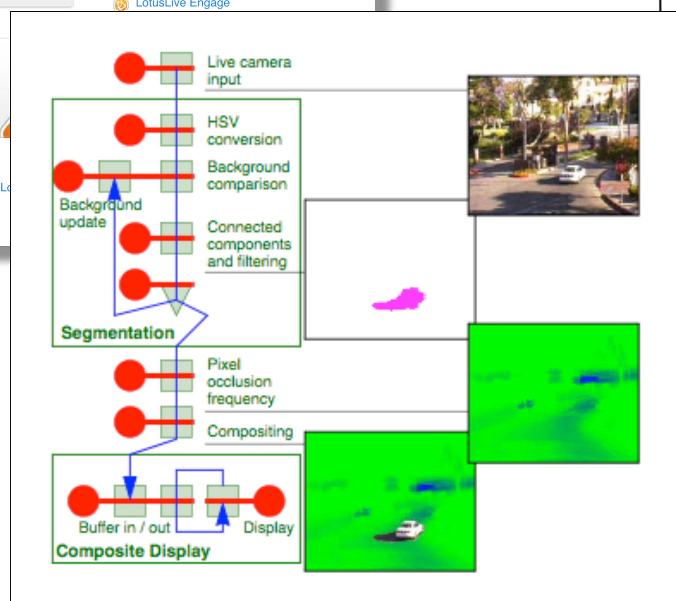


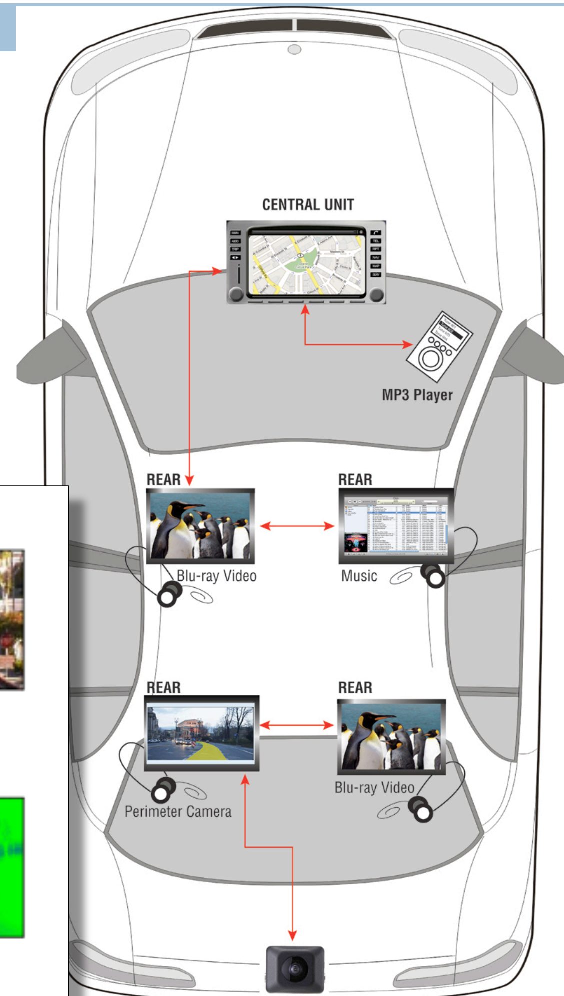
- Actors
  - Carlos Varela
- Thorn
  - Bard Bloom
  - Brian Burg
  - Jakob Dam
  - Julian Dolby
  - Nate Nystrom
  - Johan Östlund
  - Gregor Richards
  - Ignacio Solla Paula
  - Rok Strniša
  - Emina Torlak
  - Tobias Wrigstad
  - Jan Vitek

# What do these apps have in common?





# Common threads



- collection of distributed components…
- …loosely coupled by messages, persistent data
- irregular concurrency, driven by real-world data (“reactive”)
- high data volumes
- fault-tolerance important

# Why are systems distributed?



- *access to other administrative domains* with proprietary data and data processing capabilities
- *sharing* data among multiple users or administrative domains

- *scalability* via networked compute and storage resources
- *isolation* for fault containment
- *redundancy* (data or compute) for handling network partition or node failures
- *reduced latency* by bringing computation closer to human users or physical devices that access it

*Distributed apps are now the norm*

*How should our programming models adapt to this new reality?*

*Why is this interesting/challenging?*

# Distributed systems...back in the day

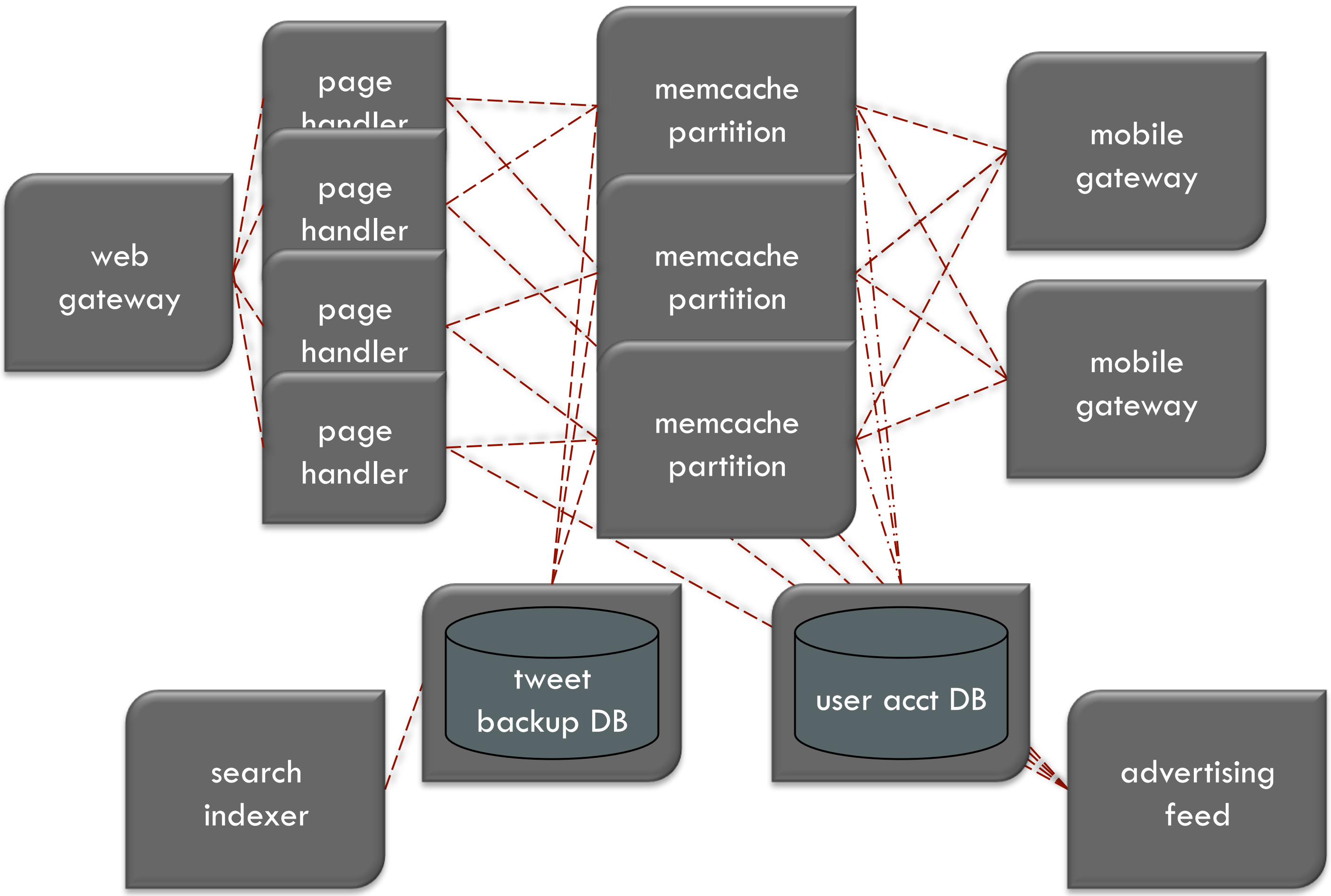


- clear distinction between "clients" and "servers"
- servers implemented standard services
  - database queries
  - NFS file access
  - FTP
  - simple HTTP requests
  - ...
- most sophisticated code on "server" side
  - e.g., for clustering
  - inter-node code written mostly by systems gurus
- application-specific APIs to access standard services

# Contrast with...



## Twitter and similar "web2.0" applications



# Distributed systems today



> A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable
>
> *Leslie Lamport*

- complex network of interconnected services
- variety of availability/reliability requirements
- distinction between "client" and "server" increasingly unclear
- many administrative domains...
- ...not all of them are your friends

# Failures have consequences

*eCommerce ca. 2002:*

Wanted: 2 different pairs of kid's sneakers from namelesswebsite.com



**Error 500**
**An error has occured while processing request:https://www.namelesswebsite.com/ErrorReporter**
**Message: Server caught unhandled exception from servlet [JSP 1.1 Processor]: null**

**Target Servlet:** JSP 1.1 Processor
**StackTrace: Root Error-1:**
java.lang.NullPointerException
    at Proxy._eProxyGetAccount_jsp_0._jspService(_eProxyGetAccount_jsp_0.java:78)
    at org.apache.jasper.runtime.HttpJspBase.service(HttpJspBase.java(Compiled Code))
    at javax.servlet.http.HttpServlet.service(HttpServlet.java(Compiled Code))
    at org.apache.jasper.runtime.JspServlet$JspServletWrapper.service(JspServlet.java(Compiled Code))
    at org.apache.jasper.runtime.JspServlet.serviceJspFile(JspServlet.java(Compiled Code))
    at org.apache.jasper.runtime.JspServlet.service(JspServlet.java(Compiled Code))
    at javax.servlet.http.HttpServlet.service(HttpServlet.java(Compiled Code))
    at com.ibm.servlet.engine.webapp.StrictServletInstance.doService(ServletManager.java(Compiled Code))
    at com.ibm.servlet.engine.webapp.StrictLifecycleServlet._service(StrictLifecycleServlet.java(Compiled Code))
    at com.ibm.servlet.engine.webapp.IdleServletState.service(StrictLifecycleServlet.java(Compiled Code))

Thank You For Your Order!
Please Visit Us Again.

# Failures have consequences

*Results*

- 3 pairs of shoes…
- …all the same
- credit card charges for 4 pairs

# Failures are e'er with us



## Twitter Backup Failure Sent the Site Crashing

*Posted on Jan 20th, 2010 by* ***Zee***

Recommend Tweet

Twitter has released an official statement regarding the outage that saw the site come to a crashing halt.

The statement:

"We are recovering from this incident. A sudden failure coupled with problems in switching to a backup system produced a high number of errors for around 90 minutes. This made the site largely inaccessible. No data was lost or compromised during this outage."

The outage was the longest downtime for some time for the Silicon Valley company, lasting almost an hour. The downtime also arrived as news of a second earthquake in Haiti broke, leading many, including ourselves, to believe the two were related.

There is also talk that Bill Gates arrival on Twitter may have caused the outage, although considering the news of his appearance broke yesterday, it's highly unlikely.

# *Composing* functionality in the presence of failures can be problematic

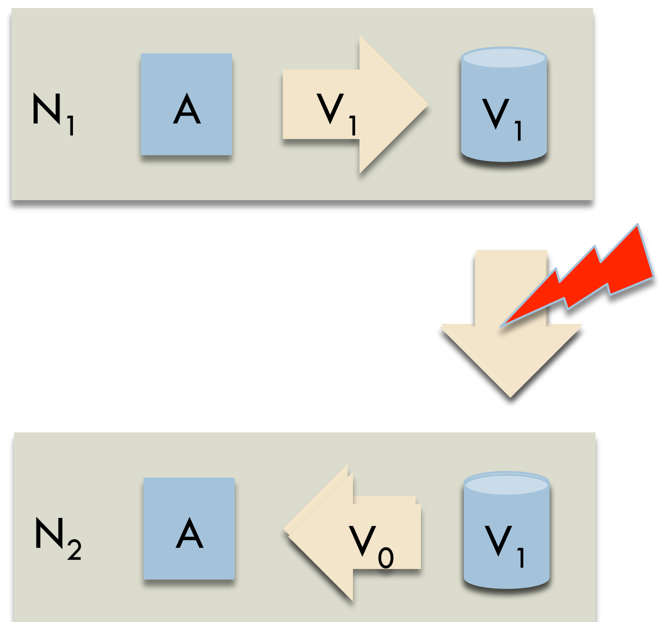


- consider:
  - composing a fast, high availability component…
  - …with a slow, fault-tolerant replicated server

# Alas, you can't have it all



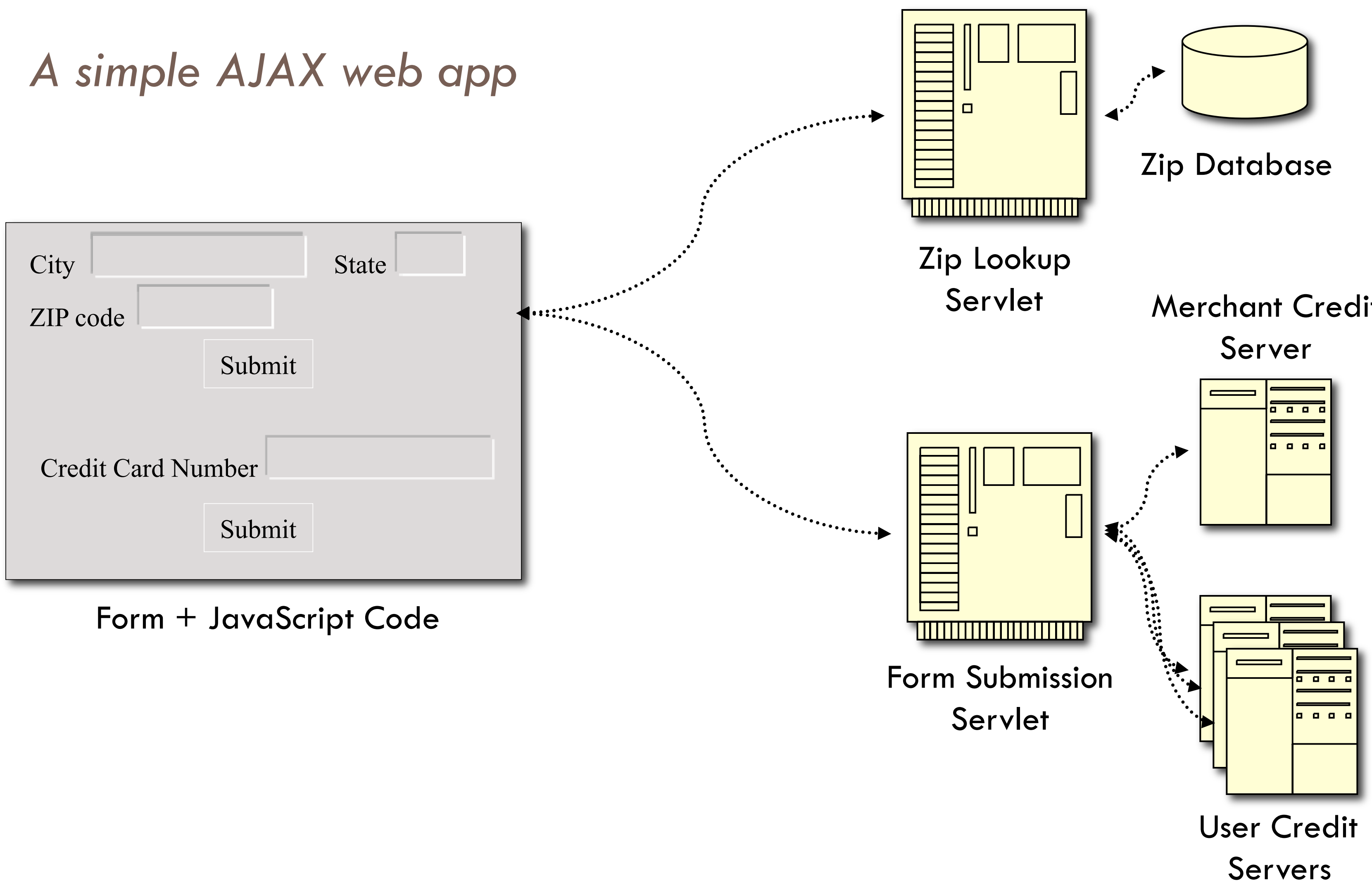
- In fact, you can only have two out of the following three*
  - consistency
  - availability
  - partition-tolerance




*Eric Brewer, *Toward Robust Distributed Systems*, 2000
(example due to Julian Browne)

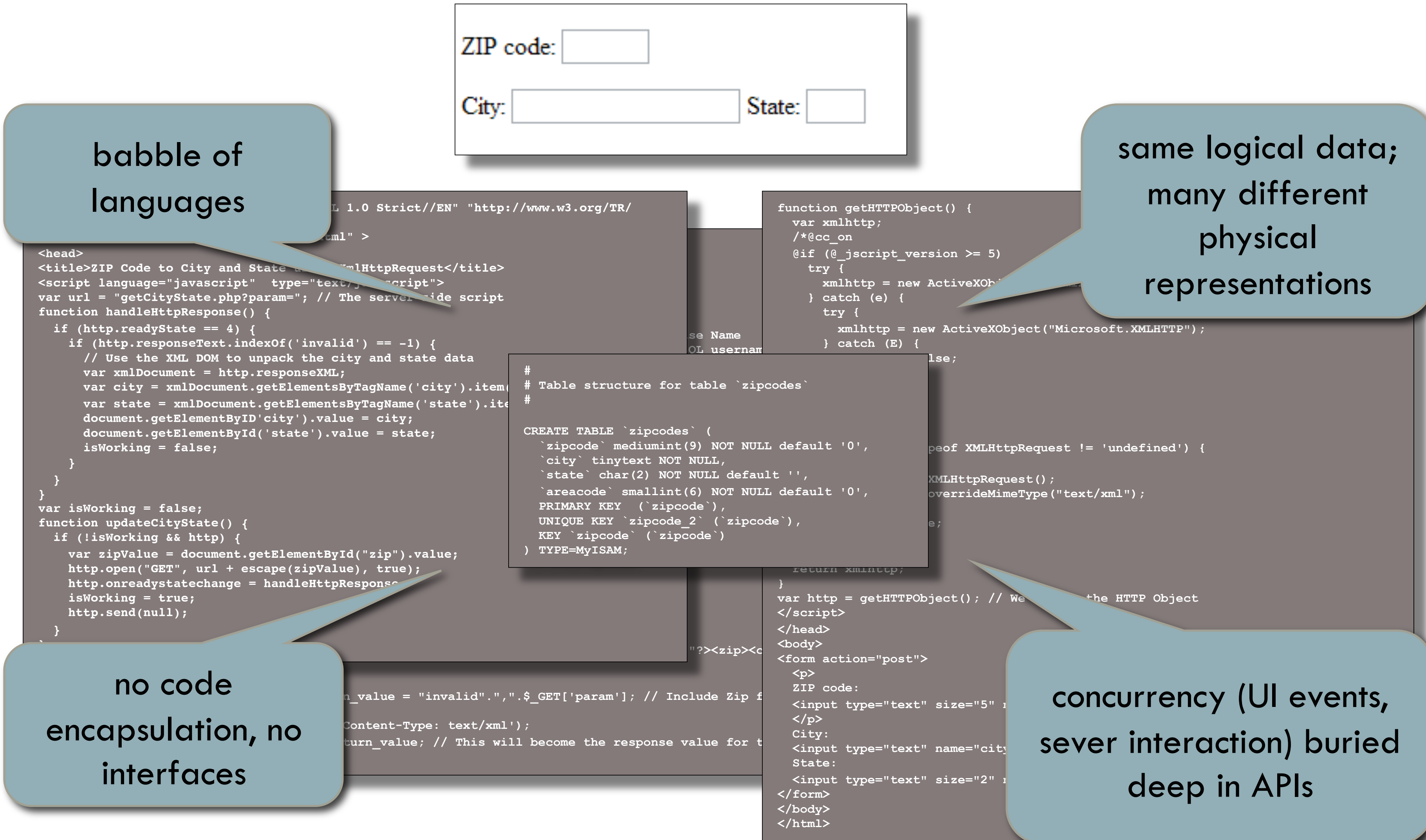
# Distributed programming can get ugly

*A simple AJAX web app*



# Code snippet for AJAX UI





# Can't we just adapt existing programming models for distribution?



problem: single address space programming concepts *cannot* be repurposed*

- latency
- identity: local vs. global
- partial failure
- ubiquitous concurrency

```
while (true) {
  try {
    table->remove(name);
    break;
  }
  catch (NotFound) {
    break;
  }
  catch (NetworkServerFailure) {
    continue;
  }
}
```

*Waldo et al., *A Note on Distributed Computing, 1994*

# What's wrong with accessing distributed services via libraries?



- problem: neither programmer nor runtime can readily reason about *composition* of components
- each library handles common distribution issues (timeouts, acknowledgments, ...) differently

# But beware of baking in too much*



- don't make developers pay for functionality they don't need
- e.g.:
  - reliable message delivery in system substrate is both redundant and expensive...
  - ...if sender of message needs acknowledgment that receiver *processed* the message correctly anyway

*Saltzer et al., *End-to-end Arguments in System Design, 1984*

# What do we want in a distributed programming model?



- allows sufficient control of low-level behavior to tune performance and reliability
- doesn't require ubiquitous, expensive functionality (end-to-end argument)
- doesn't suffer from Waldo et al's pathologies...
- ...but allows reuse of familiar programming concepts when appropriate

# Proposed way forward: *Actor* model



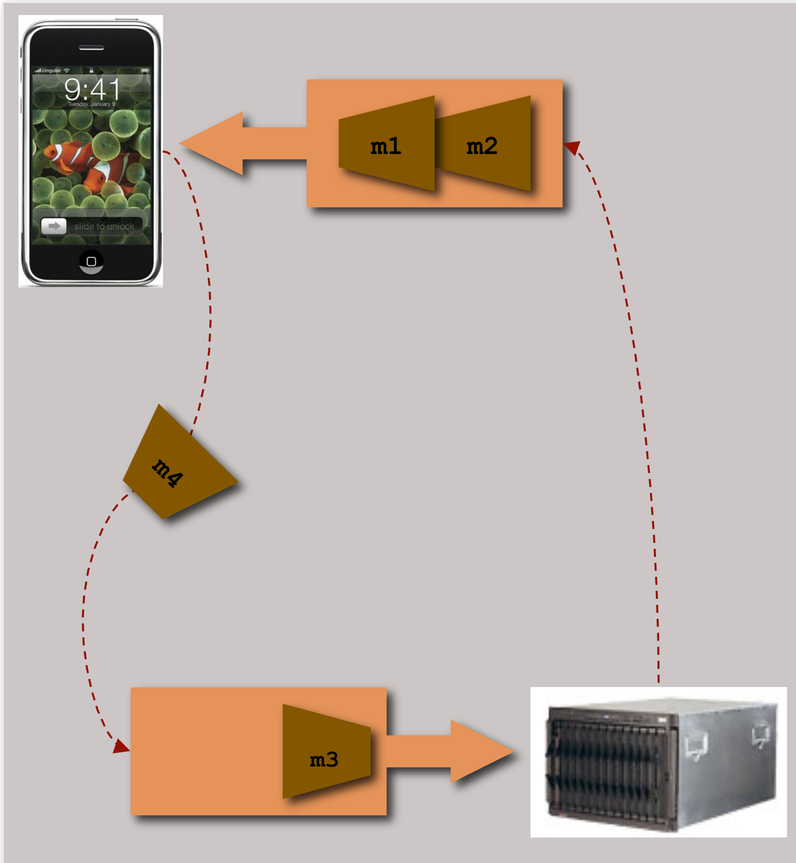
- originally defined by Hewitt et al.* in '73 to model properties of certain AI planners...
- ....then developed as a general distributed programming model by others, particularly Agha
- has gone in and out of fashion
- realized in a wide variety of languages, e.g.:
  - Erlang
  - Salsa
  - Scala
  - Axum
  - ...
- our implementation is called *Thorn*

*Hewitt et al., *A Universal Modular Actor Formalism for Artifial Intelligence, 1973*

# Actor basics



- actor is a single-threaded stateful process
- collection of actors form a program/ system
- state of one actor not (directly) accessible by another: *isolation*
- every actor has a unique *name*
- actor *names* are data
- actors communicate by sending *messages* to one another
    - messages sent *asynchronously*: sender does not block awaiting receipt
    - actor names may be sent as messages
- received messages managed by a (conceptually unbounded) *mailbox*
    - no message ordering guarantee
- in response to a message, an actor may:
    - update its state
    - create new actors (and remember their names)
    - send messages



# Actor variants



- mechanisms for updating state
  - functional (state passed as continuation between messages)
  - imperative (state explicitly mutated between messages)
- message delivery may or may not be guaranteed
- explicit "peeking" into mailbox may or may not be allowed
- explicit or implicit message receipt
- infinite behaviors (e.g., sending unbounded numbers of messages) may or may not be allowed
- ordered or unordered (implicitly concurrent) actions on message receipt

# Actor and distribution



- actor topologies are highly dynamic
  - communication topology is dynamic, since names may be sent as messages
  - set of actors can grow dynamically via creation
- asynchronous messaging allows behaviors of sender and receiver to be decoupled
- actors are oblivious to locality
  - but actors running on same node, or same address space amenable to many optimizations
- concurrency
  - data races are impossible
  - messsage waiting deadlocks are possible, but arise via poor protocol design, not unfortunate scheduling decisions

# Our actor language: *Thorn*



*An open source, agile, high performance language for concurrent/distributed applications and reactive systems*

## Key research directions

- *code evolution:* language, runtime, tool support for transition from prototype scripts to robust apps
- *efficient compilation:* for a dynamic language on a JVM
- *cloud-level optimizations:* high-level optimizations in a distributed environment
- *security:* end-to-end security in a distributed setting
- *fault-tolerance:* provide features that help programmers write robust code in the presence of hardware/software faults

# Features, present and absent

*Features*

- isolated, concurrent, communicating processes
- lightweight objects
- first-class functions
- explicit state...
- ...but many functional features
- powerful aggregate datatypes
- expressive pattern matching
- dynamic typing
- lightweight module system
- JVM implementation and Java interoperability
- gradual typing system (experimental)

*Non-features*

- changing fields/methods of objects on the fly
- introspection/reflection
- serialization of mutable objects/ references or unknown classes
- dynamic code loading

# Thorn status



- Open source: http://www.thorn-lang.org
- Interpreter for full language
- JVM compiler for language core
  - no sophisticated optimizations
  - performance comparable to Python
  - currently being re-engineered
- Initial experience
  - web apps, concurrent kernels, compiler, ...
- Prototype of (optional) type annotation system

# Simple Thorn script



access command-line args

file i/o methods

split string into list

```
for (l <- argv()(0).file().contents().split("\n"))
  if (l.contains?(argv()(1))) println(l);
```

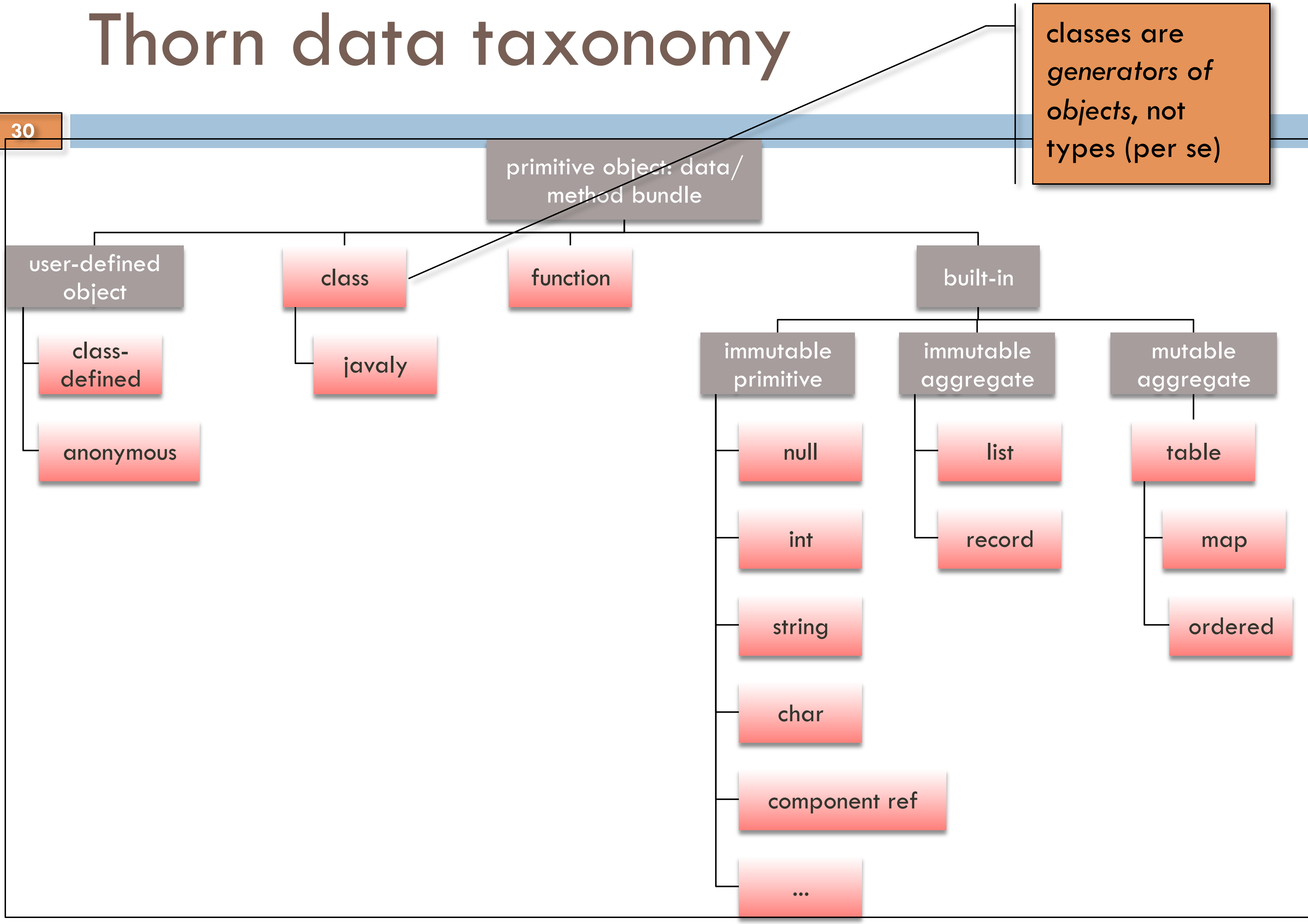
iterate over elements of a list

no explicit decl needed for var

usual library functions on lists

# DEMO

# Thorn data taxonomy



# Thorn features for more robust scripting



- no reflection, eval, dynamic code loading
  - alternatives for most scenarios
- ubiquitous patterns
  - for documentation
  - to generate efficient code
- powerful aggregates
  - allow semantics-aware optimizations
- easy upgrade path from simple scripts to reusable code
  - simple records → encapsulated classes
- modules
  - easy to wrap scripts, hide names
- experimental gradual typing system

# A MMORPG*

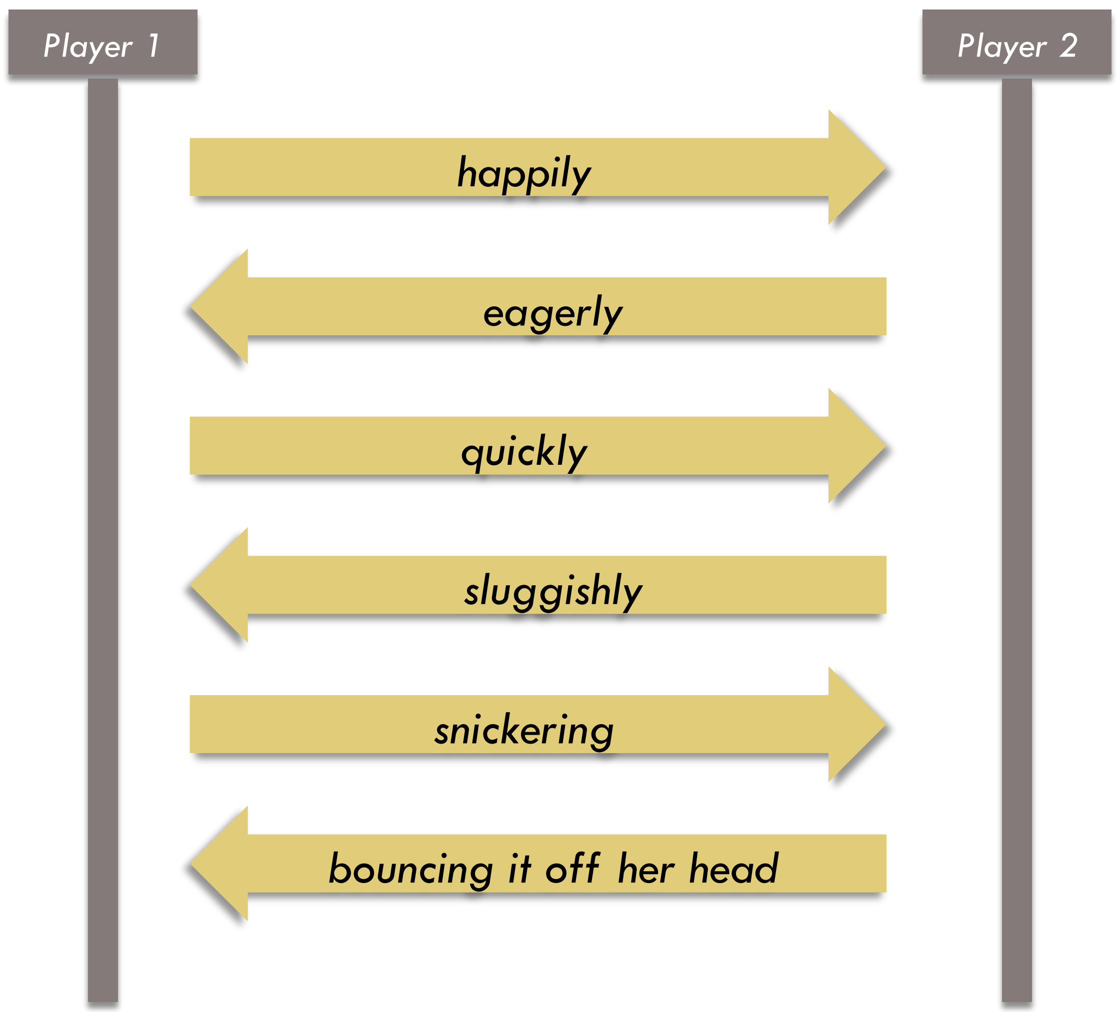


- adverbial ping-pong
- two players
- play by describing how you hit the ball
- distributed
- each player runs exactly the same code

*minimalist multiplayer online role-playing game

# MMORPG message flow

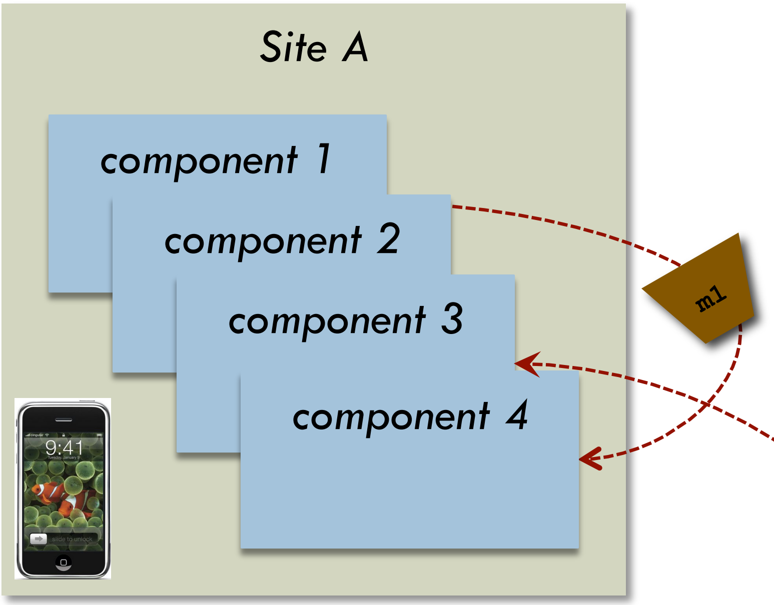




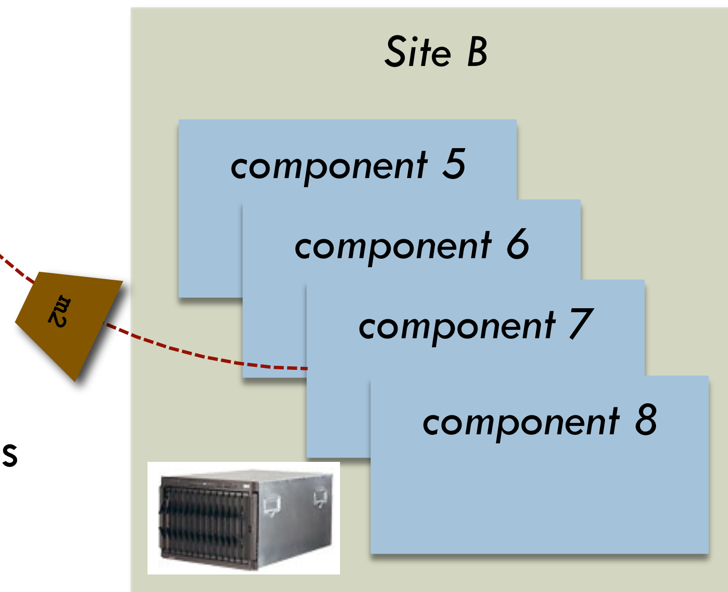
# DEMO

MMORPG

# Thorn refines actors with *sites*




- *components* are Thorn processes
- components can spawn other components (at the same site)
- processes communicate by message passing
- intra- and inter-site messaging *works the same way*



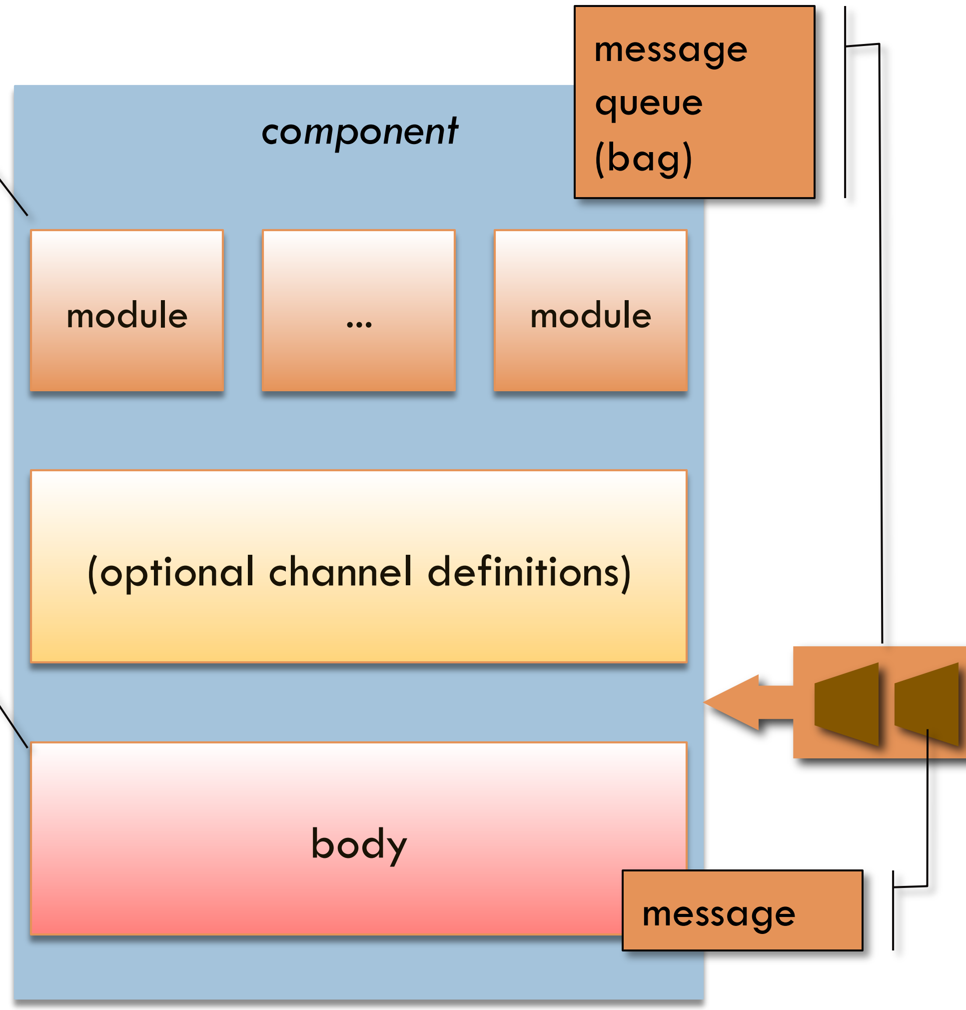

- *sites* model physical application distribution (implemented as one JVM per site)
- I/O and other resources managed by sites
- failures managed by sites
- components can be spawned at remote sites
- optimizations for intra-site messaging, concurrency

# Anatomy of a component



- defines the component's code and state
- loaded and initialized when component is spawned

- statement executed when component is spawned (usually a loop)
- component execution ends when body ends



# MMORPG

```
// MMORPG code for both player

spawn {

  var done := false;

  body {
    [name, otherURI] = argv();
    otherSite = site(otherURI);

    fun play(hit) {
      advly = readln("Hit how?");
      done := advly == "";
      if (done) {
        println("You lose!");
        otherSite <<< null;
      }
      else {
        otherSite <<<
        "$name $`hit`s the ball $advly.";
      }
    }

    start =
      thisSite().str < otherSite.str;

    if (start) play("serve");

    do {
      receive {
        msg:string => {
          println(msg);
          play("return");
        }
      | null => {
          println("You
          done := true;
        }
      }
    } until (done);
  }
```

spawn an isolated *component* (process)

mutable component-scoped variable

immutable component-scoped variable

convert URI into component ref

function decl

receive messages matching *pattern*

send a message (any immutable datum)

pattern variable (with type constraint)

constant pattern

interpolate data into string

# Thorn design philosophy



- steal good ideas from everywhere
  - (ok, we invented some too)
  - aiming for harmonious merge of features
  - strongest influences: Erlang, Python (but there are many others)

- assume concurrency is ubiquitous
  - this affects *every* aspect of the language design

- adopt best ideas from scripting world…
  - dynamic typing, powerful aggregates, …

- …but seduce programmers to good software engineering
  - powerful constructs that provide immediate value
  - optional features for robustness
  - encourage use of functional features when appropriate
  - no reflective or self-modifying constructs

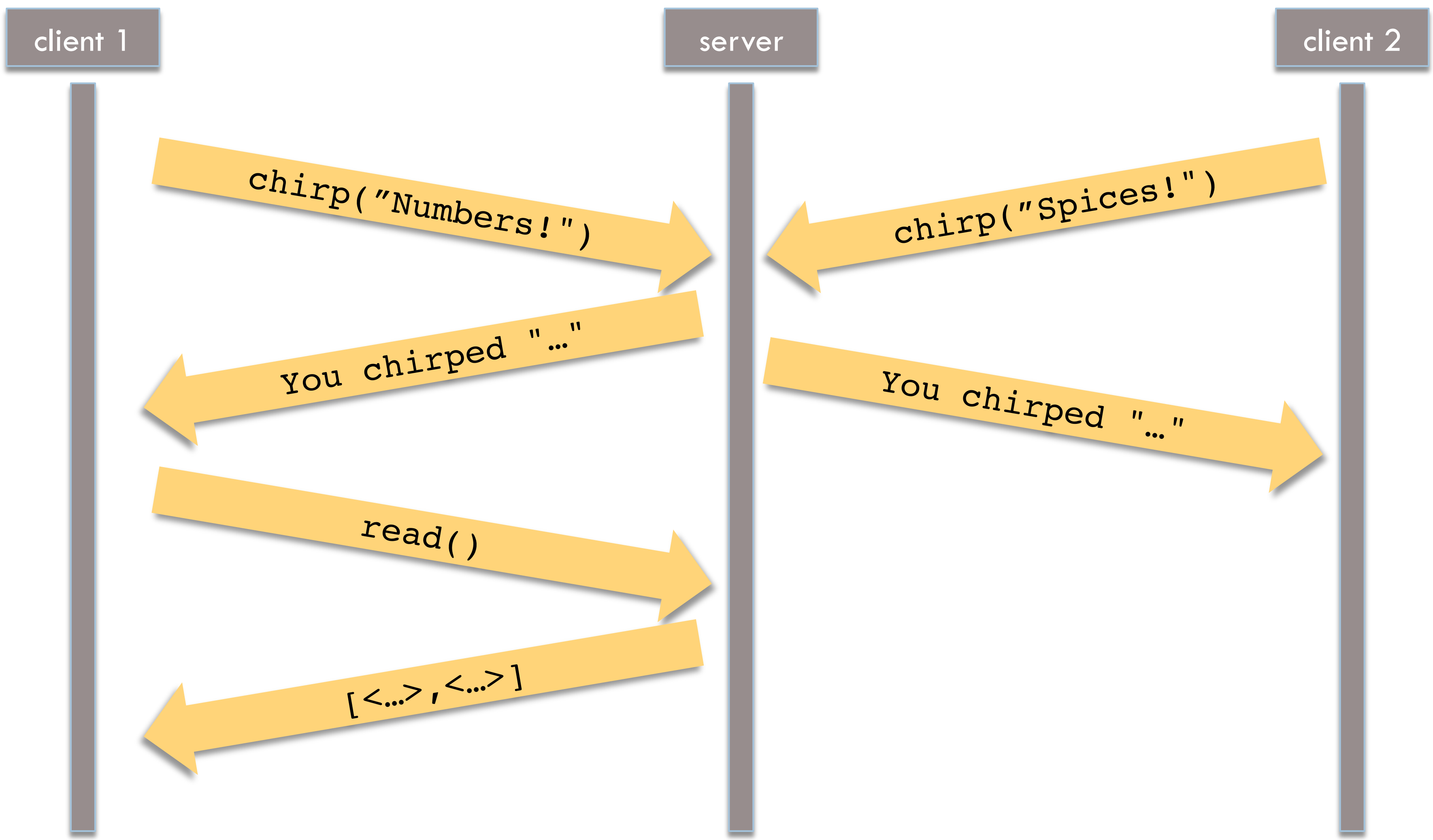
# Scripting + concurrency: ? …or… !



- scripts already handle concurrency (but not especially well)
- dynamic typing allows code for distributed components to evolve independently…code can bend without breaking
- rich collection of built-in datatypes allows components with minimal advance knowledge of one another's information schemas to communicate readily
- powerful aggregate datatypes extremely handy for managing component state
  - associative datatypes allow distinct components to maintain differing "views" of same logical data

# Cheeper: Twitter in a few lines of code





# Cheeper client code



```
spawn chclient {
import CHEEPER.*;
server = site(argv()(0));

fun help() {
 println("? = help");
 println("/ = read");
 println("+N = vote for");
 println("-N = vote against");
 println("other = chirp that");
 }

fun read() {
 c's = server <-> read();
 for( <chirp, plus, minus>  <- c's) {
   println(
     "$chirp [+$plus/-$minus]");
   }
 }
```

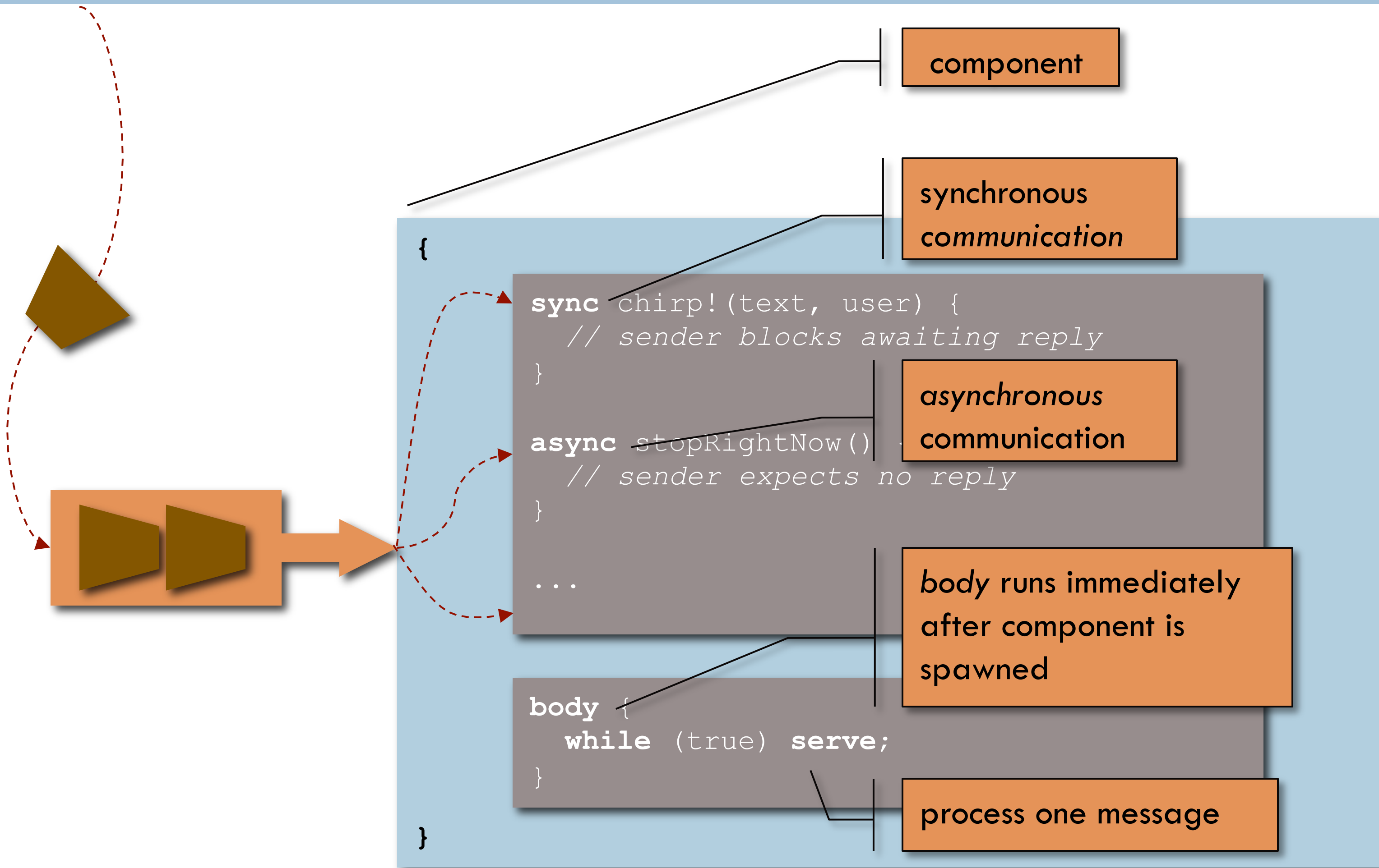
```
body {
 println("Welcome to Cheeper!");
 println("? for help");

 user = readln("Who are you? ");
 while(true) {
   s = readln("Chirp: ");
   match(s)
     "?" => help()
   | "/" => read()
   | "\\+([0-9]+)" / [.int(n)] =>
     println( server <-> vote(n, true))
   | "\\-([0-9]+)" / [.int(n)] =>
     println(server <-> vote(n, false))
   | _ =>
     println(server <-> chirp!(s,user))
 }
}
}
```

# Cheeper server code

```
spawn chserver {
import CHEEPER.*;

users = table(user)<var chirps>;
chirps = table(n)<chirp, var plus, minus>;

sync chirp!(text, user){
 n = chirps.num;
 c = Chirp(text,user,n);
 chirps(n) :=
   < chirp=c,
     plus=0,
     minus=0 >;
 if (users.has?(user))
   users(user).chirps ::= c
 else
   users(user) := < chirps=[c] >;
 "You chirped '$c'"
 }

fun love(<plus, minus>) = plus - minus;
```

```
sync read() =
  sort[row
    incrby love(row)
    decrby chirp.n
  | for row && <chirp> <- chirps];

sync vote(n, plus?) {
 if (plus?)
   chirps(n).plus += 1
 else
   chirps(n).minus += 1;
 "Thanks"
 }

body{
 println("Cheeper server here!");
 while(true) {
   println("Server ready...");
   serve;
 }
}
}
```

# Augmenting basic actors with channel-style communication




Channels are sugar on basic actor primitives

# Channel-style communication



- server defines communications:

  ```
  sync chirp!(text,user) { ... }
  ```

  - RPC

  ```
  async stopRightNow() from $(root) prio 100 {...}
  ```

  - signal
- client can call these

  ```
  response = server <-> chirp!("Hey!","Me")
  ```

  ```
  server <-- stopRightNow()
  ```

  - timeout option available on <->
- server determines when channels are interrogated

  ```
  serve // respond to one communication
  ```

  - ... timeout / administrative options.

# Further actor extensions for Thorn: work in progress (I)



- local coordination: *chords*
  - pattern on *multiple* mailbox messages
  - inspired by join calculus, polyphonic C#
- local checkpoint/recovery
  - sites can recognize failed components
  - certain variables designated as *stable*; written through to stable storage on every write
  - `init` and `reinit` code blocks in component
    - `init` establishes component invariants when component starts
    - `reinit` re-establishes invariants from stable variables after a crash

# Further actor extensions for Thorn: work in progress (II)



- data access
  - *remote table:* hybrid of table and component
  - queries shipped to same site of remote table, executed in own component
- capability-style security
  - component as unit of trust, isolation
  - piggyback on messaging

# Actors vs. design desiderata



*Waldo et al.*

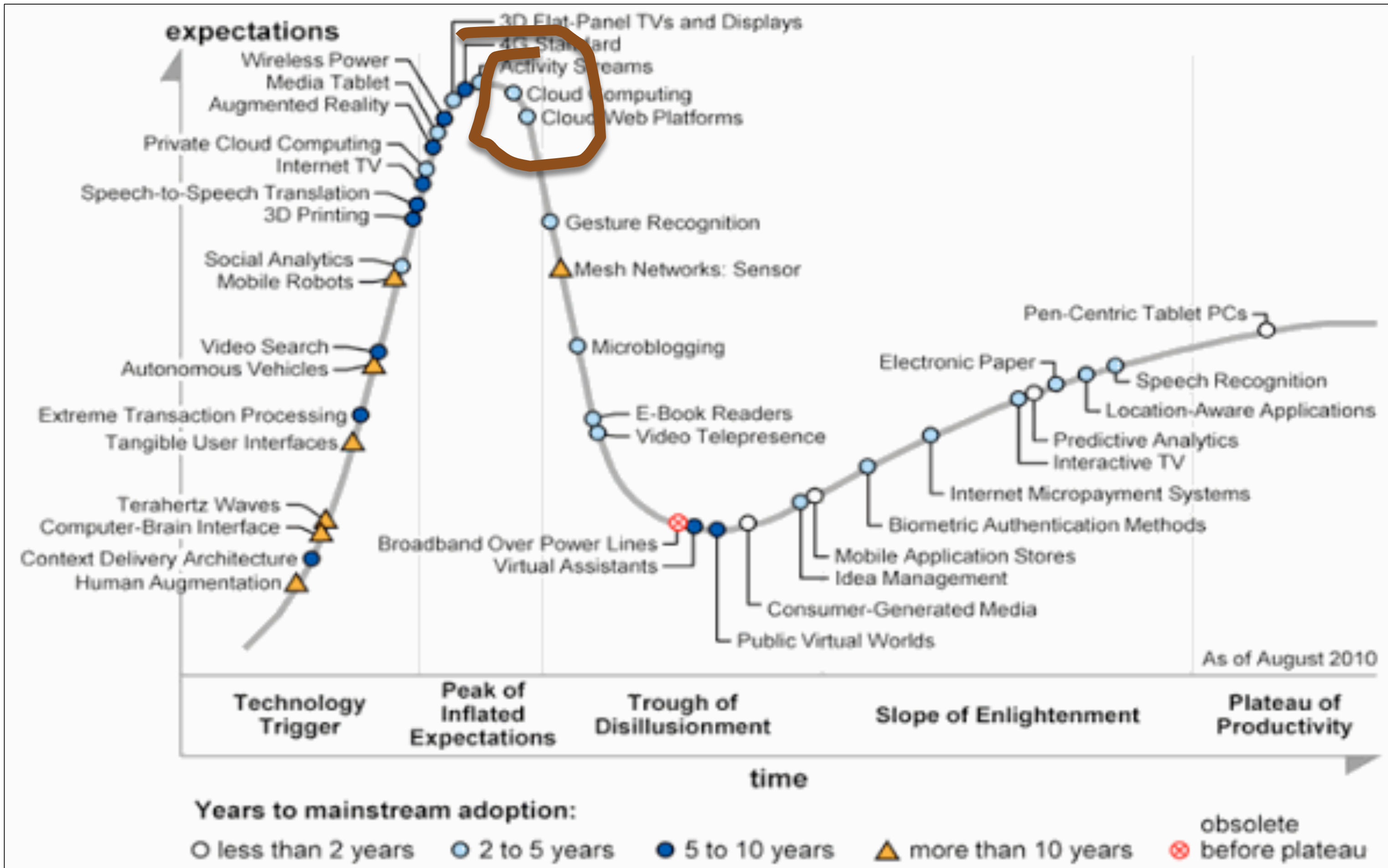
- latency?
  - explicit distinction between cheap local operations and potentially expensive remote ones
- identity?
  - only notion of global identity is actor name
- ubiquitous concurrency?
  - actors are inherently concurrent
- partial failure?
  - distinction between local operations and remote messages is helpful
  - original actor model assumed guaranteed message delivery; Thorn does not
  - original model made no assumptions about node failure; Thorn assumes possible

*Saltzer et al*

- are core features useful and cost-effective?
  - composition via name passing cheap and natural for the internet
  - asynchronous messaging is cheap and unavoidable
  - ability to dynamically spawn actors is necessary for topology to evolve, and can be made cheap

# Cloud computing: state of the hype*






*Gartner Group, 2010

# Is there something really new here?



## Environmental factors

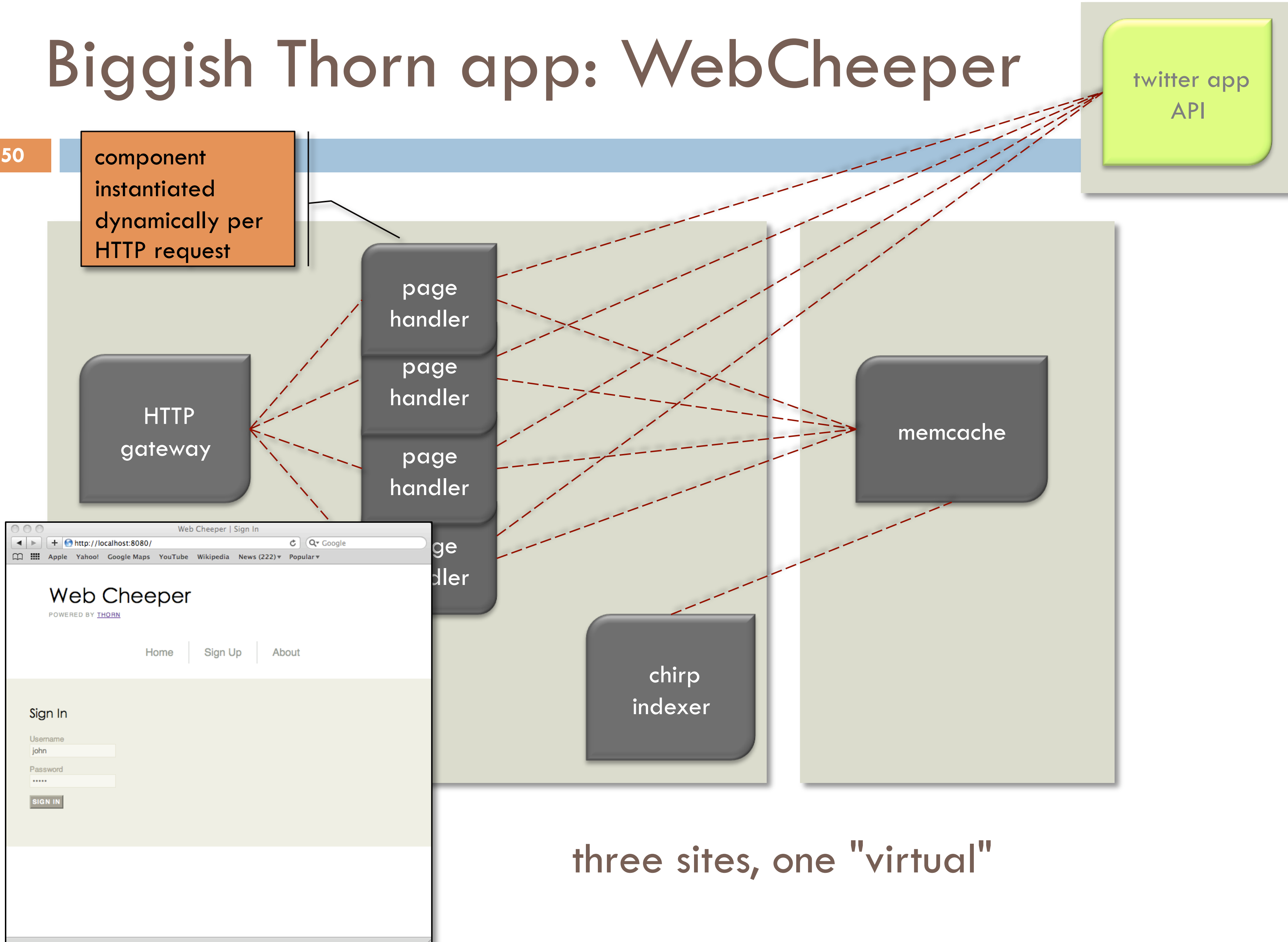
- increasing disconnect between hardware and software platforms
  - virtual hardware, virtual language runtimes, portable middleware
- ubiquitous network connectivity
  - comfort with data/computation "somewhere else"
- high-quality web UIs
  - browser as universal GUI for remote apps
- cost of wide-area networking has fallen more slowly than other IT hardware costs
  - economic necessity mandates putting the data near the application [Gray, 2003]

## New functionality

- *managed* collection of (relatively) uniform distributed resources
- the illusion of infinite computing resources available on demand
  - scaling down as important as scaling up

# Biggish Thorn app: WebCheeper

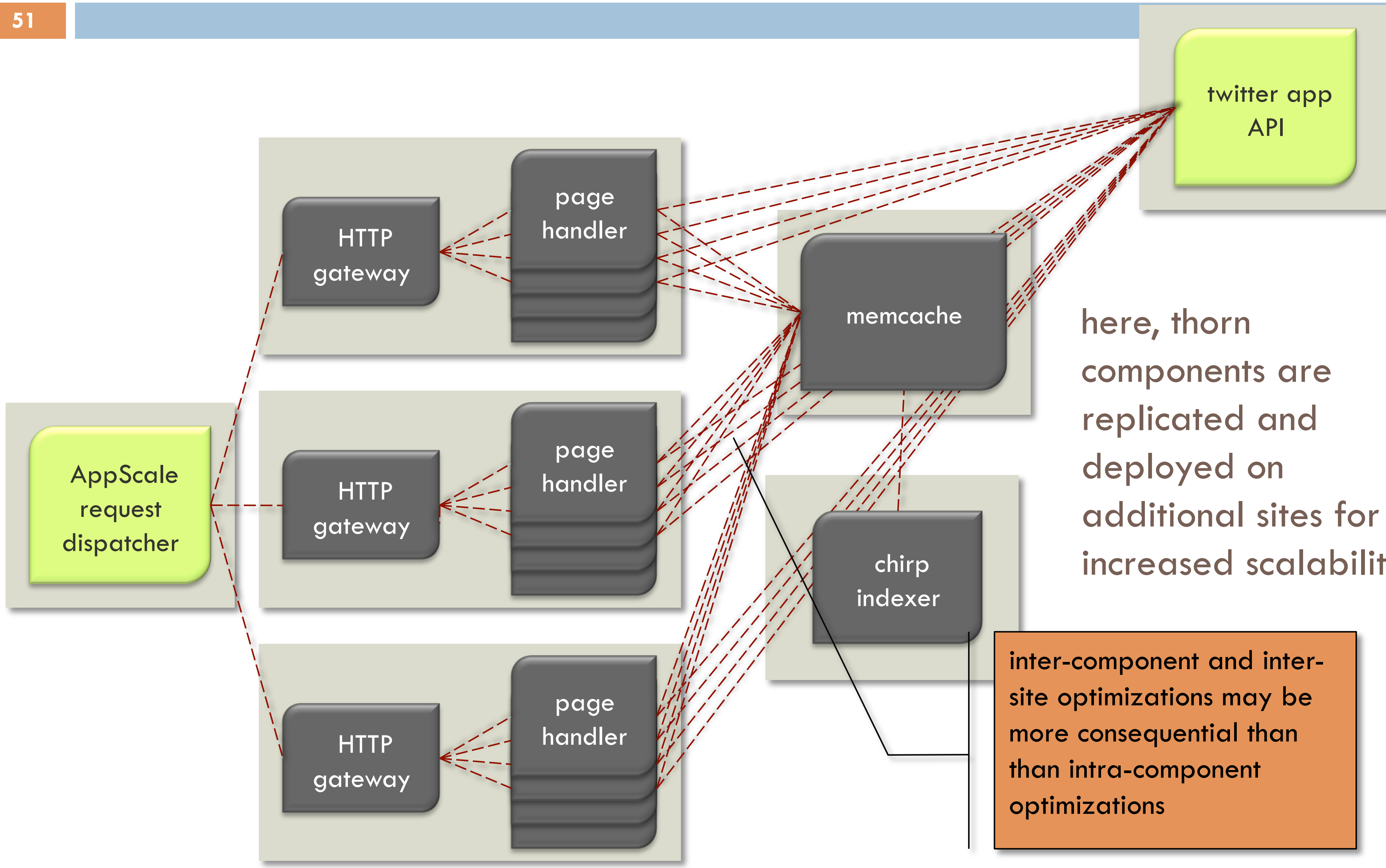





three sites, one "virtual"

# WebCheeper on AppScale cloud






here, thorn components are replicated and deployed on additional sites for increased scalability

inter-component and inter-site optimizations may be more consequential than than intra-component optimizations

# Replication: key to scalability and fault-tolerance



- replicated compute servers
- replicated databases
- caching throughout the internet
- splitting disjoint data, disjoint services over multiple nodes

# Opportunity: recomposing actors for cloud optimization I



- simple data splitting
  - split components whose communications access disjoint data
- replicate *stateless* components
  - as in WebCheeper example
  - can arbitrarily replication components where state not accessed across multiple communications
- speculative replication of stateful components
  - when downstream peers are *idempotent* w.r.t. repeated requests
- *sharding*
  - split components with table state into multiple components, multiple tables with disjoint key spaces
  - possible when component accesses only a single table record

# Opportunity: recomposing actors for cloud optimization II



- batch→stream
  - replace pipeline of bulk data transformations with parallel per-item transformations
- generalized map-reduce
  - identify parallelizable queries, break into pipelines
- caching
  - introduce intermediate components that store the results of computations
- weak consistency replicated datastores (à la Amazon Dynamo, Google BigTable)
  - are they an instance of a more general paradigm?

# *Transactor* model: global checkpointing



- in addition to basic actor operations, a transactor *t* can:
  - *stabilize*: enter a mode where *t* does not change its state (a non-stable transactor is *volatile*)
  - *checkpoint*: create a persistent copy of current state (restored after restart from failure)
    - checkpoint only allowed if *t* and transactors on which *t* depends are stable
    - *t* becomes volatile after checkpoint
  - *rollback*: revert to *t*'s last checkpointed state
- semantics maintains *dependence* information about peer transactors

*Field, Varela 2005

# Summary



- actors are good match for Waldo and Saltzer's desiderata
- thorn: pragmatic extension/interpretation of actor model
  - no assumption of message delivery
  - site/component distinction
  - explicitly imperative local computation
  - channels as well as simple messages
  - unbounded behaviors
- for the future: need more compositional tools
  - that enable analysis of latency, failure modes
  - enable CAP tradeoffs
  - optimization through replication

# Thanks!

Questions?